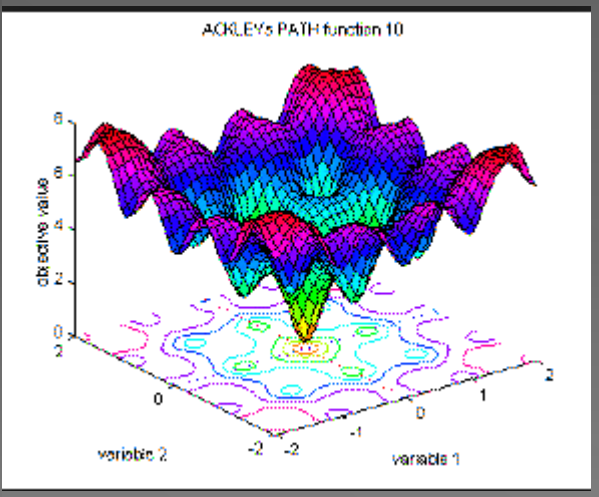
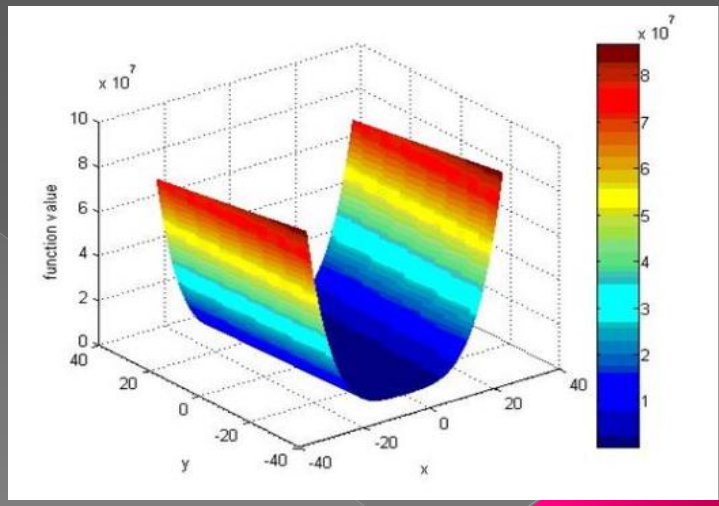
Solving Optimization Problems in Python

Lecturer: Sławomir
Presentation: Mahboobeh





Optimization Problems

- Finding the best values of weights in NNs
- Finding the best settings for a controller
- Finding a good strategy in a game

Optimization Problems

- Continuous optimization
- Real numbers with bounds: constrained optimization
- Integers: integer programming
- combinations of the above

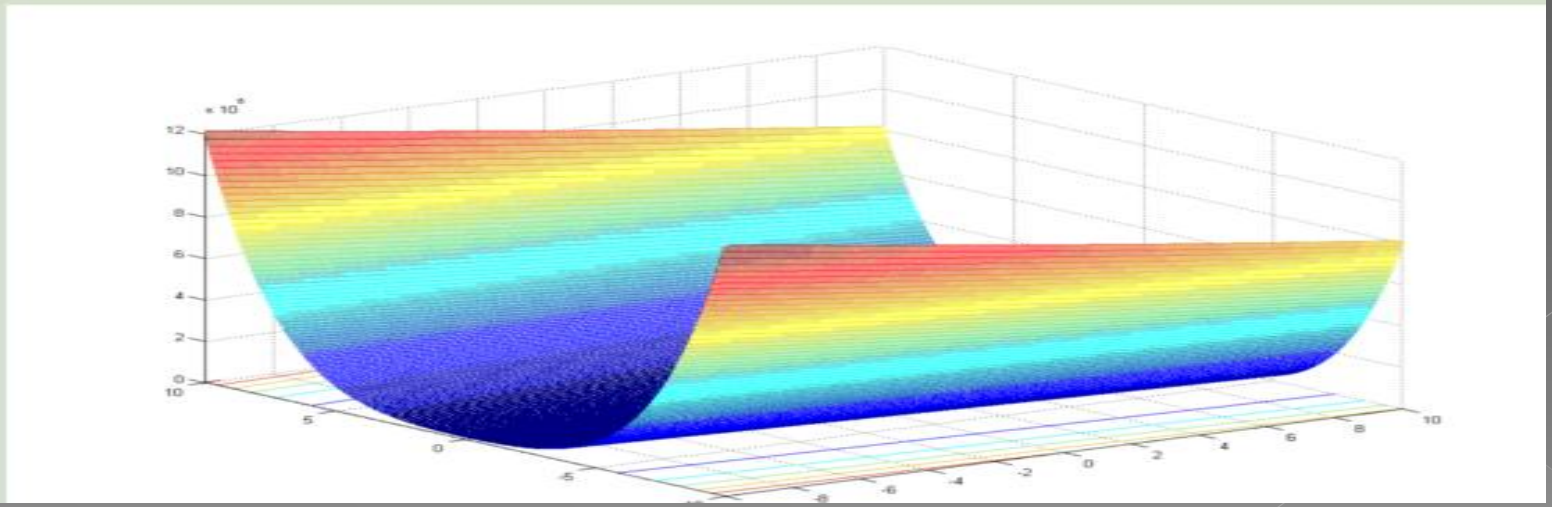
Rosenbrock Function

Rosenbrock Function

- Number of variables: n variables.
- Definition:

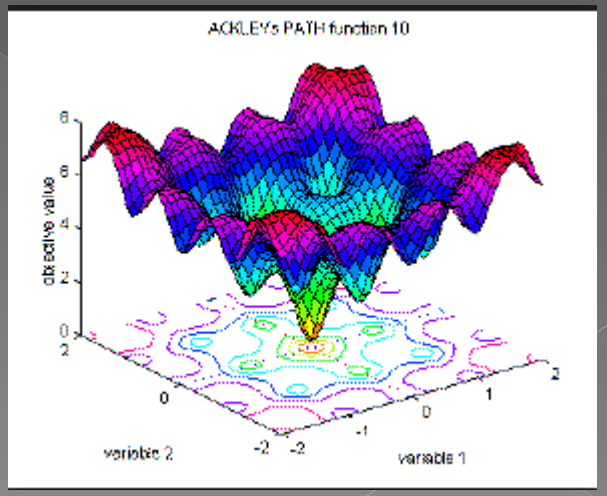
$$f(\mathbf{x}) = \sum_{i=1}^{n-1} \left[100 \left(x_i^2 - x_{i+1} \right)^2 + \left(x_i - 1 \right)^2 \right].$$

- Search domain: $-5 \leq x_i \leq 10,\ i = 1, 2, \ldots, n.$
- Number of local minima: several local minima.
- The global minima: $\mathbf{x}^* = (1, \ldots, 1), f(\mathbf{x}^*) = 0.$
- MATLAB Code: download rosen.m
- Function graph: for $n = 2$.



Ackly Function

- It is an n-dimensional highly multimodal function that has a large number of local minima but only one global minimum.
- It is a typical problem to solve with evolutionary algorithms. The function has a global minimum at the origin x = 0 with value 0.0.



Libraries in Python

- **Scipy**: http://**docs.scipy.org/doc/scipy/reference/tutorial/optimize.html**
- **ECsPy** (Evolutionary Computations in Python)-Bio-inspired Computation
- **Aima-python**: http://code.google.com/p/aima-python/
- **Pagmo**:http://pagmo.sourceforge.net/pygmo/
- **PyBrain**: http://pybrain.org/

Scipy

- The scipy.optimize package provides several commonly used optimization algorithms.
- The module contains:
- Unconstrained and constrained minimization of multivariate scalar functions (minimize) using a variety of algorithms (e.g. BFGS, Nelder-Mead simplex, Newton Conjugate Gradient, COBYLA or SLSQP)
- Global (brute-force) optimization routines (e.g., anneal, basinhopping)
- Least-squares minimization (leastsq) and curve fitting (curve_fit) algorithms
- Scalar univariate functions minimizers (minimize_scalar) and root finders (newton)
- Multivariate equation system solvers (root) using a variety of algorithms (e.g. hybrid Powell, Levenberg-Marquardt or large-scale methods such as Newton-Krylov).

Scipy(MINIMIZE)

- Import :

```
>>> import numpy as np
>>> from scipy.optimize import minimize
```

- Defining Function :

```
>>> def rosen(x):
...     """The Rosenbrock function"""
...     return sum(100.0*(x[1:]-x[:-1]**2.0)**2.0 + (1-x[:-1])**2.0)
```

```
>>> x0 = np.array([1.3, 0.7, 0.8, 1.9, 1.2])
>>> res = minimize(rosen, x0, method='nelder-mead',
...                options={'xtol': 1e-8, 'disp': True})
```

ECsPy (Evolutionary Computations in Python)

- **Evolutionary computation,**
- **Swarm intelligence,**
- **Neural networks.**

Evolutionary Computations

- Encoding problem to the components of evolutionary methods: **genetic algorithm**, **ant colony optimization**, **particle swarm optimization**.
- Applying evolutionary operators
- Selecting new generation for next iteration

Evaluationary Optimization for Rosenbrock Function

- Import libraries

```
from random import Random
from time import time
import inspyred
```

- Defining Problem and representing a binary encoding

```
problem = inspyred.benchmarks.Binary(inspyred.benchmarks.Rosenbrock(2),
                                     dimension_bits=50)
```

```
problem = inspyred.benchmarks.Rosenbrock(2)
```

ec – Evolutionary computation framework

- It provides a framework for creating evolutionary computations
- **inspyred.ec.Bounder**(*lower_bound=None*, *upper_bound=None*)
- i**nspyred.ec.DiscreteBounder**(*values*)
- i**nspyred.ec.Individual**(*candidate=None*, *maximize=True*)
- i**nspyred.ec.EvolutionaryComputation**(*random*)

Evolutionary Algorithm

- **GA**: inspyred.ec.GA
- **ACO**: inspyred.swarm.ACS
- **PSO:** inspyred.swarm.PSO
- **EvolutionaryComputation:** ea.evolve

```
final_pop = ea.evolve(generator=problem.generator,
                      evaluator=problem.evaluator,
                      pop_size=50,
                      maximize=problem.maximize,
                      bounder=problem.bounder,
                      max_evaluations=300000,
                      num_elites=1)
```

inspyred.ec.EvolutionaryComputation

- Represents a basic evolutionary computation.
- This class encapsulates the components of a generic evolutionary computation:
- the selection mechanism,
- the variation operators,
- the replacement mechanism,
- the migration scheme,
- the archival mechanism,
- the terminators,
- and the observers.

inspyred.ec.GA(*random*)

- This class represents a genetic algorithm which uses, by default, rank selection, *n*-point crossover, bit-flip mutation, and generational replacement. In the case of bit-flip mutation, it is expected that each candidate solution is a Sequence of binary values.
- Optional keyword arguments in evolve args parameter:
- ***num_selected*** – the number of individuals to be selected (default len(population))
- ***crossover_rate*** – the rate at which crossover is performed (default 1.0)
- ***num_crossover_points*** – the *n* crossover points used (default 1)
- ***mutation_rate –*** the rate at which mutation is performed (default 0.1)
- ***num_elites*** – number of elites to consider (default 0)

Example#1(GA for Rosenbrock)

```
from random import Random
from time import time
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Binary(inspyred.benchmarks.Rosenbrock(2),
                                         dimension_bits=50)
    ea = inspyred.ec.GA(prng)
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=50,
                          maximize=problem.maximize,
                          bounder=problem.bounder,
                          max_evaluations=300000,
                          num_elites=1)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea

if __name__ == '__main__':
    main(display=True)
```

ea.terminator

- **Algorithm termination methods**
- This module provides pre-defined terminators for evolutionary computations.
- Terminators specify when the evolutionary process should end. All terminators must return a Boolean value where True implies that the evolution should end.
- All terminator functions have the following arguments:
- ***population* –** the population of Individuals
- ***num_generations*** – the number of elapsed generations
- ***num_evaluations*** – the number of candidate solution evaluations

Example#1(PSO for Rosenbrock)

```
from time import time
from random import Random
import inspyred

def main(prng=None, display=False):
    if prng is None:
        prng = Random()
        prng.seed(time())

    problem = inspyred.benchmarks.Rosenbrock(2)
    ea = inspyred.swarm.PSO(prng)
    ea.terminator = inspyred.ec.terminators.evaluation_termination
    ea.topology = inspyred.swarm.topologies.ring_topology
    final_pop = ea.evolve(generator=problem.generator,
                          evaluator=problem.evaluator,
                          pop_size=10,
                          bounder=problem.bounder,
                          maximize=problem.maximize,
                          max_evaluations=30000,
                          neighborhood_size=5)

    if display:
        best = max(final_pop)
        print('Best Solution: \n{0}'.format(str(best)))
    return ea

if __name__ == '__main__':
    main(display=True)
```

inspyred.swarm.topologies.ring_topology

This module defines various topologies for swarm intelligence algorithms. It determines the logical relationships among particles in the swarm (i.e., which ones belong to the same“neighborhood”).

- **inspyred.swarm.topologies.ring_topology(*random*, *population*, *args*)**
- **inspyred.swarm.topologies.star_topology(*random*, *population*, *args*)**

It has the following arguments:

- *random* – the random number generator object
- *population* – the population of Particles

PyBrain (Black-box)

General Black-box optimizers

class `pybrain.optimization.`**`RandomSearch`**(*evaluator=None*, *initEvaluable=None*, ***kwargs*)

Every point is chosen randomly, independently of all previous ones.

class `pybrain.optimization.`**`HillClimber`**(*evaluator=None*, *initEvaluable=None*, ***kwargs*)

The simplest kind of stochastic search: hill-climbing in the fitness landscape.

class `pybrain.optimization.`**`StochasticHillClimber`**(*evaluator=None*, *initEvaluable=None*, ***kwargs*) ¶

Stochastic hill-climbing always moves to a better point, but may also go to a worse point with a probability that decreases with increasing drop in fitness (and depends on a temperature parameter).

`temperature`

The larger the temperature, the more explorative (less greedy) it behaves.

PyBrain (Black-box)

Continuous optimizers

class `pybrain.optimization.`**`NelderMead`**(*evaluator=None*, *initEvaluable=None*, ***kwargs*)

Do the optimization using a simple wrapper for scipy's fmin.

class `pybrain.optimization.`**`CMAES`**(*evaluator=None*, *initEvaluable=None*, ***kwargs*)

CMA-ES: Evolution Strategy with Covariance Matrix Adaptation for nonlinear function minimization. This code is a close transcription of the provided matlab code.

class `pybrain.optimization.`**`OriginalNES`**(*evaluator=None*, *initEvaluable=None*, ***kwargs*)

Reference implementation of the original Natural Evolution Strategies algorithm (CEC-2008).

class `pybrain.optimization.`**`ExactNES`**(*evaluator=None*, *initEvaluable=None*, ***kwargs*)

A new version of NES, using the exact instead of the approximate Fisher Information Matrix, as well as a number of other improvements. (GECCO 2009).

`baselineType`

Type of baseline. The most robust one is also the default.

class `pybrain.optimization.`**`FEM`**(*evaluator=None*, *initEvaluable=None*, ***kwargs*)

Fitness Expectation-Maximization (PPSN 2008).

Thank You!